Verifying security protocols using theorem provers(Algebras, Languages, Computations and their Applications)

AUTHOR(S):

Tanaka, Miki

CITATION:

Tanaka, Miki. Verifying security protocols using theorem provers(Algebras, Languages, Computations and their Applications). 数理解析研究所講究録 2007, 1562: 79-86

ISSUE DATE:

2007-06

URL:

http://hdl.handle.net/2433/81103

数理解析研究所講究録
第1562巻 2007年 79-86

Verifying security protocols using theorem provers

Miki Tanaka
National Institute of Information and Communications Technology
Koganei, Tokyo 184-8795, Japan
Email: miki.tanaka@nict.go.jp

Abstract

This is a report on a project in formal verification of cryptographic protocol. We adapt Paulson's work of verification of TLS (Transport Security Layer) on Isabelle/HOL [1] to the proof-assistant Coq. Paulson's trace model of TLS is translated into the Coq syntax and the proofs of security properties are explicitly constructed on the Coq system.

1 Introduction

It has been understood that getting security proofs right is very subtle and difficult. Hence, formal verification of security, employing so called formal method technology has become a recent challenge in the field. The idea is to use some strictly defined language (formal language) to model and reason about the system of which you want to verify the properties. Here "strictly defined" in effect means that the language and the reasoning mechanism are syntactically explicitly defined so that the reasoning process can be rigidly executed by computers. Therefore, if one can establish a proof, in a formal manner using such techniques, that a security system has a certain security property, then, it is a good quality guarantee of security for the system, modulo the assumptions with the modelling. Of course one can argue what counts as a "good quality guarantee", but at least here, there is no room for human errors that come to light after many years in the process of machine reasoning.

Application of such technology to providing security guarantees for cryptographic protocols has been attempted for a while by now, starting from around the discovery of an attack to the Needham-Schroeder Public Key protocol by Lowe using a model-checker in 1995. Since then there have been various different techniques in formal method at varying level in abstraction applied to security proofs. Among those techniques we are interested in automated theorem proving because it allows for realistic modelling constraints and for extensibility with verification that takes computational complexity aspects into account.

We report on a study on Paulson's formal verification of TLS (Transport Layer Security) using Isabelle [1]. We adapted Paulson's work to the proof assitant Coq, by translating the formal model into the Coq syntax and construct-

ing the proof on Coq system. In the following we take a quick look at both the method and the result: after reviewing the notion of automated theorem proving itself, we quickly go though the syntax, or equivalently, the language, for describing protocol execution on theorem provers. We don't give the precise definition of the syntax but it is the same as Paulson's syntax, presented in the style of Coq. Then we briefly explain some of the security properties of TLS that are proved by Paulson and also give their representations in Coq.

2 Automated Theorem Proving

Automated theorem proving is a research for using computers to reason in formal languages based on various logics. Given that mathematical logic can be viewed as pure symbolic manipulation, we can expect that computers can go quite far with it, although we cannot expect them to provide a complete proof for all possible valid statements

In this report we are interested in systems that use expressive higher order logics, and in such cases, the systems cannot be fully automated. Instead they are often "interactive", indicating that the users need to provide guides to the system when proving theorems. To what extent human involvement is required differs depending on the system.

Paulson used interactive theorem prover Isabelle with higher order logic to prove some security property of the protocol Transport Layer Security [1]. Both Isabelle and Coq are in principle an LCF-style generic theorem prover. Isabelle is equipped with relatively strong automation, and is originally written by Paulson and others [2]. The automation is based on the two parts called "simplifier" (the rewriting engine) and "classical reasoner" (based on the tableaux method). Coq is based on an intuitionistic type theory called Calculus of Indutive Construction and allows one to extract certified programs from proofs.

3 Verification of Transport Layer Security

Paulson used interactive theorem prover Isabelle with higher order logic to prove some security property of the protocol Transport Layer Security [1].

3.1 TLS

TLS (Transport Layer Security, formerly called SSL, Secure Socket Layer) is a protocol that provides secure connection above transport protocols such as TCP and below application protocols such as HTTP, SMTP, etc. Currently, TLS is mostly used in combination with those application protocols and form HTTPS, etc., to provide the secure connection version of those protocols. As it is used extensibly in many activities by common users over the Internet, it has been standardized by IETF as RFC2246 [4] first time in 1999 (version 1.0). Originally it was developed by Netscape as SSL from around 1996 and the current version is TLSv1.1.

TLS protocol starts with a phase called "handshake" protocol. This phase is for negotiation between the client and server on the encryption algorithms and so on (cipher suite) to be used and establishing session keys using public-key based key exchange, before actually starting to send encrypted messages over the connection. Some certificate issuing authority is typically assumed for the public key based key exchange, providing authentication only for the server.

3.1.1 TLS Handshake

Our main concern is the handshake part of TLS since this is the part of the TLS protocol Paulson modelled using Isabelle. In this section, we briefly explain a simplified version of this protocol as in [1]. Formal models are constructed based on this version.

The client starts the negotiation by sending the ClientHello message containing his own name, a nonce Na, sessionID, and a cipher suite that the client wants to use in the following communication. Receiving this, the server responds with ServerHello message which contains another nonce Nb, sessionID and another cipher suite of his own choice. Following ServerHello, the server also sends to the client his own public key certificate signed by the CA. On receiving these, the client generate another nonce called "pre-master secret (PMS)" and send it to the server encrypted by the public key of the server (ClientKeyExchange). Only the server should be able to decrypt this message, and hence, at this point, the client and server share three nonces Na, Nb, PMS, and PMS is a shared secret. From these three, the both parties compute the "master secret (MS)" using some pseudo-random function. From this master secret both the client and the server compute session keys to be used to encrypt subsequent communication. As confirmation, each sends Finished messages, which consist of all the preceding messages exchanged between them encrypted with the session key just established.

3.2 Modelling Protocols

In order to model protocol execution syntactically, we take a view that communication protocol executions consist of sequences of message exchanges and some related incidents (or, say, actions). We model such actions as "events" and the execution of protocols are modelled by the notion of "traces", which are defined as sequences (lists) of events. Then the security properties we would like to prove become those of the set of traces that are associated to the particular protocol of interest. Since lists are one of the most basic inductively defined datatypes, the proofs of properties of traces are basically done by induction on lists.

3.2.1 A quick look on the syntax

The syntax consists of the following datatypes: agents, message, event, key.

Agents, Keys, Messages The datatype agent has three constructors. `Spy` and `Server` to represent the assumed adversary and the assumed CA who signs public key certificates. The third constructor `Friend` takes a natural number as the index to represent unbounded number of ordinary participants.

```
Inductive agent := Spy | Server | Friend : nat → agent.
```

Agents are divided into two groups: whether they belong to the subset `bad` or not. We assume two axioms stating that `Spy` ∈ `bad` and `Server` ∉ `bad`. For `Friend i`, in Coq it is also necessary to state explicitly that for any natural number `i` either `Friend i` ∈ `bad` or `Friend i` ∉ `bad` holds. The private keys of agents in `bad` are at `Spy`'s disposal.

The datatype keys are defined indirectly using functions. Keys for encryption and those for decryption are associated using the `invKey` function. Public keys are defined as `publicKey a` for any agent a, and private keys are defined by composition: `privateKey a = invKey (publicKey a)`. The two kinds of session keys, `clientK` and `serverK`, which are derived after successful execution of handshake, are defined using the function `sessionK` which takes three natural numbers (nonces) and a flag indicating whether it is for the client or the server. The fact that any session key `k` is a symmetric key is stated as `invKey k = k`. We also need to state the injectivity of these functions so that we can concluded the equality of the arguments from the equality of the values.

The datatype message is defined inductively: constructors `Agent` and `Key` are for coercing data of type agent and key to those of type message. Similarly, `Nonce` and `Number` are for natural numbers. `Hash` and `MPair` take one or two messages, respectively, to produce another message, which is the hash and the pair (notation {|`x`, `y`|}) of the arguments. Finally `Crypt` takes a key and a message to produce a message, which is the encrypted message using the key.

Events The datatype `event` corresponds to execution steps of protocols. An execution of the protocol is modelled as a list of events, which are called "traces". We only need part of Paulson's theory Event for the verification of TLS. We distinguish two different kinds of event: `Says` a b x, where a b are agents and x is a message, represents the event when agent a sends message x to agent b. The second kind, `Notes` a x represents the event that agent a internally take note of message x.

Functions on message sets In the analysis, several functions on messages are used: the function `analz` takes a set of messages and returns the set of messages one can obtain by decomposing pairs and decrypting using the keys contained in that set. This is the ability one expects for active adversaries in terms of retrieving information from a set of known messages. The function `parts` is similar but the decryption is obtained for free, providing the set of submessages for the given set of messages. The third function `synth` is for composing new messages: it takes a set of messages as argument and returns the set of messages one can compose from the original set and the publicly available information using taking pairs and hashes and encrypting using available keys.

Initial States and knowledge sets The knowledge of agents during the course of protocol execution is described using the inductively defined function `knows`. Given a list of events evs and an agent a, `knows a evs` returns the set of messages that agent a can obtain by observing the trace evs. At the start of execution, all agents have associated initial states of their knowledge. For each `Friend i` and `Server`, it consists of the set of all public keys of participating agents and his own private key. For `Spy`, in addition to above, he also knows the set of private keys of bad agents. The function `spies` is defined as `knows Spy`.

3.2.2 TLS traces

Using the language defined in the previous section, each specific protocol is modelled as a set of rules for generating traces that correspond to runs of the protocol. For the case of TLS, the following inductive definition defines the set of TLS traces. Here only a few out of 15 rules are shown.

```
Inductive tls : Ensemble (list event) :=
  | tls_nil : In _ tls nil

  | tls_fake : ∀ (evs:list event)(x:message)(a:agent),
        In _ tls evs →
        In _ (synth (analz (spies initState evs))) x →
        In _ tls ((Says Spy a x)::evs)

  | tls_spyKeys : ∀ (evs:list event)(na nb m:nat)(r:role),
        In _ tls evs →
        In _ (analz (spies initState evs)) (Nonce na) →
        In _ (analz (spies initState evs)) (Nonce nb) →
        In _ (analz (spies initState evs)) (Nonce m)  →
        In _ tls ((Notes Spy (MPair (Nonce (PRF m na nb))
                                    (Key (sessionK na nb m r))))::evs)
...
```

`tls_nil` is the rule for the base case stating that an empty list is a tls trace. The next two rules, `tls_fake` and `tls_spyKeys` are for describing `Spy`'s possible behaviour. `tls_fake` states that `Spy` can say whatever message he can synthesize and analyse from messages in his the knowledge he builds up during observing the protocol run. `tls_spyKeys` states that when `Spy` knows three nonces, he can use them to create a session key.

A new session in the handshake protocol is initiated by application of the following rule:

```
...
  | tls_clientHello : ∀ (evs:list event)(na pa sid:nat)(a b:agent),
        In _ tls evs → ¬ In _ (used evs) (Nonce na) →
        ¬ (∃ x, ∃ y, ∃ z, PRF x y z = na) →
        In _ tls ((Says a b
                 {| (Agent a), (Nonce na), (Number sid), (Number pa) |})::evs)
...
```

An agent (client) should use a fresh nonce that is not in the range of `PRF` to create a correct message to send to the other party (server).

The rule that plays the central role in the TLS handshake protocol is this:

```
...
| tls_clientKeyExch : ∀ (evs:list event)(pms:nat)(a b b':agent)(kb:key),
        In _ tls evs → ¬ In _ (used evs) (Nonce pms) →
        ¬ (∃ x, ∃ y, ∃ z, PRF x y z = pms) →
        List.In (Says b' a (certificate b kb)) evs →
        In _ tls ((Says a b (Crypt kb (Nonce pms))) ::
                              (Notes a {| Agent b, Nonce pms |}) ::evs)
...
```

After the client and the server finished their hello messages, the client creates another nonce `pms` and uses server's public key `kb` to send it securely. After application of this rule, the both party now have a shared secret `pms`.

3.3 Properties Proved

Paulson categorises the properties he proved in several criteria and discusses them in detail in [1]. We consider that what he calls security theorems are the final goals of analysis. We proved in Coq all those security theorems and most of other properties Paulson proved in his TLS theory script. In this section, we present some of those security theorems using Coq syntax in order to give the flavour of the properties formally proved in Paulson's model.

The property `analz_insert_key` states that the loss of a session key won't allow the `Spy` to learn more nonces that he should not know.

```
Lemma analz_insert_key : ∀ (evs:list event)(x y z n:nat)(r:role),
  In _ tls evs →
  (In _ (analz (Add _ (spies initState evs) (Key (sessionK x y z r))))
        (Nonce n))
  ↔ (In _ (analz (spies initState evs)) (Nonce n)).
```

`analz_insert_key` is proved by application of a useful lemma:

```
Lemma analz_image_keys : ∀ (evs:list event)(n:nat),
  In _ tls evs →
  ∀ K:Ensemble key, (∀ k, In _ K k →
                          ∃ x, ∃ y, ∃ z, ∃ r,
                                        sessionK x y z r = k) →
  (In _ (analz (Union _ (Im _ _ K Key) (spies initState evs))) (Nonce n))
  ↔ (In _ (analz (spies initState evs)) (Nonce n)).
```

This lemma states that, even if you add a set of session keys to `Spy`'s knowledge, it does not let `Spy` to learn any more nonce. The proof of `analz_image_keys` is quite complex.

Another set of examples we present here is about `pms` and `ms`.

```
Lemma Spy_not_see_pms :
  ∀ (evs:list event)(pms:nat)(a b:agent),
  List.In (Notes a {| Agent b, Nonce pms |}) evs →
  In _ tls evs →
  ¬ In _ bad a → ¬ In _ bad b →
  ¬ In _ (analz (spies initState evs)) (Nonce pms).
```

This lemma states that when agents a and b are not bad and the rule ClientKeyExch was applied between them and a shared secret pms is established, then pms is not derivable from the Spy's current knowledge. This lemma is used in proofs of other lemmas. In particular, it is used in the similar lemma about the master secret ms

```
Lemma Spy_not_see_ms :
  ∀ (evs:list event)(pms na nb:nat)(a b:agent),
  List.In (Notes a {| Agent b, Nonce pms |}) evs →
  In _ tls evs →
  ¬ In _ bad a → ¬ In _ bad b →
  ¬ In _ (analz (spies initState evs)) (Nonce (PRF pms na nb)).
```

The proofs of both lemmas are quite substantial.

4 Proof in Coq

Adapting the proof scripts in Isabelle to Coq was not a simple task. This mainly due to the fact that Isabelle is equipped with an efficient term rewriting engine and proof search engine that allow reasonable automation, while Coq is more focused on constructive proofs and proof checking and is equipped with some automation tactics. At the moment, what we have proved are the lemmas given by Paulson, but still, we consider this experience to be a valuable one. Paulson's original Isabelle proof scripts are well adapted for maximising the performance of Isabelle's simplifier and the reasoner; the lemmas are formulated in such a way that the reasoner and the simplifier can make most use of them. However, those formulations were not the best for constructing more explicit proofs in Coq. So we reformulated many of the lemmas and, as far as proving TLS properties go, we could prove them without the help of many of lemmas Paulson prepared to be used by the simplifier. Our current scripts (consisting of Message.v, Event.v, Public.v and tls.v) have 161 lemmas proved in total. Paulson's Isabelle scripts have more than 270 lemmas, although simple comparison is not appropriate here because the Coq scripts do not contain some lemmas present in Isabelle scripts that are not relevant for proving TLS properties. For many cases where the proofs in Isabelle were just the combination of "blast" and a few lines of codes for treating special subgoals, we had to explicitly construct proofs using induction on the construction of TLS traces, which would normally produce 15 subgoals to start with. On the other hand, we could dispense with many lemmas that are rather simple and seem to exist to help automation, because they are often not very usable in the explicit proofs we need to provide with Coq. Again, the numbers are only for a reference but the Coq scripts contain around 6500 lines of codes, while those for Isabelle scripts add up to around 2600.

As for the automation in Coq, we added to the Hint database all the constructors from the syntax, and some axioms stating several decision properties. The basic strategy for most proofs is induction on the structure of TLS traces, often combined with the case analysis for an agent being bad or not.

5 Conclusion

We have used the proof-assistant Coq to reformulate Paulson's modelling and formal security proof of Transport Layer Security. Although two theorem provers share a lot in common in principle, their usage and their mechanisms differ substantially, and hence, the resulting proofs are also very different. Our proofs are more explicitly constructed than those done in Isabelle due to the fact that the level of automation is different with Coq. The proof script consists of four files as is the case for Isabelle. We will soon make public the Coq proof scripts adapted to V8.1.

Acknowledgment

This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Japan (MEXT), Grant-in-Aid for Young Scientists (B), Grant number 187602935003.

References

[1] L. Paulson, "Inductive analysis of the internet protocol TLS", ACM Transactions on Information and System Security, vol.2, No.3, pages 332–351, 1999.

[2] T. Nipkow, L. Paulson, M. Wenzel, "Isabelle/HOL: a proof assistant for higher-order logic", Lecture Notes in Computer Science, Vol. 2283, 2002.

[3] The Coq proof assistant, `http://coq.inria.fr`.

[4] T. Dierks and C. Allen, "The TLS Protocol", IETF, RFC 2246, January 1999. `http://www.ietf.org/rfc/rfc2246.txt`.